KAAPI: A thread scheduling runtime system for data flow computations on cluster of multi-processors

Thierry Gautier, Xavier Besseron, Laurent Pigeon
INRIA, projet MOAIS
LIG, Bâtiment ENSIMAG
51 avenue Jean-Kuntzmann
F-38330 Montbonnot St Martin, France
thierry.gautier@inrialpes.fr, xavier.besseron@imag.fr, laurent.pigeon@imag.fr

ABSTRACT

The high availability of multiprocessor clusters for computer science seems to be very attractive to the engineer because, at a first level, such computers aggregate high performances. Nevertheless, obtaining peak performances on irregular applications such as computer algebra problems remains a challenging problem. The delay to access memory is non uniform and the irregularity of computations requires to use scheduling algorithms in order to automatically balance the workload among the processors.

This paper focuses on the runtime support implementation to exploit with great efficiency the computation resources of a multiprocessor cluster. The originality of our approach relies on the implementation of an efficient work-stealing algorithm for a macro data flow computation based on minor extension of POSIX thread interface.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*distributed programming, parallel programming*

General Terms

Performance, Algorithms, Experimentation

Keywords

Work-stealing, dataflow, multi-core, multi-processor, cluster

1. INTRODUCTION

Multithreaded languages have been proposed as a general approach to model dynamic, unstructured parallelism. They include data parallel ones – e.g. NESL [4] –, data flow – ID [9] –, macro dataflow – Athapascan [15] –, languages with fork-join based constructs – Cilk [7] – or with additional synchronization primitives – Jade [34], EARTH [19] –.

Efficient execution of a multithreaded computation on a parallel computer relies on the schedule of the threads among the processors. In the work-stealing scheduling [7], each processor gets its own stack implemented by a deque of frames. When a thread is created, a new frame is initialized and pushed at the bottom of the deque. When a processor become idle, it tries to steal work from a victim processor with a non-empty deque; then it takes the topmost frame (the least recently pushed frame) and places it in its own deque which is linked to the victim deque.

Such scheduling has been proven to be efficient for *fully-strict* multithreaded computations [7, 12] while requiring a bounded memory space with respect to a depth first sequential execution [8, 29].

In Cilk [7], the implementation is highly optimized and follow the *work first principle* that aims at moving most of the scheduling cost of from the work of the execution to the rare case during work-stealing operations. Lock free algorithm manages the access to the deque allowing to reach good efficiency even for fine grain application. Cilk is one of the best known system that allows developers to focus on the algorithm rather than the hardware. Nevertheless, Cilk targets are shared memory machines. Although a past version of Cilk, called Cilk-NOW [6], was designed for network of workstations, the current highly optimized Cilk cannot runs on multiprocessor cluster. The underlaying DAG consistency memory model remains challenging to implement for large scale clusters.

Data flow based languages do not suffer from penalty due to memory coherence protocol, mainly because data movement between instructions are explicit in the model. Since the 70s, data flow models have moved to multi-threaded architecture where key points of original model have been conserved in order to hide latency of (remote) memory access.

The EARTH [19, 28] Threaded-C language exposes two level of threads (standard threads and *fiber* threads) that allows a fine control over the effective parallelism. Fibers are executed using a data flow approach: a synchronization unit detects and schedules fibers that have all their inputs produced. In [37], the authors present a thread partitioning and scheduling algorithm to statically gather instructions into threads.

The paper focuses on the implementation of KAAPI [25], a runtime support for scheduling irregular macro data flow programs on a cluster of multi-processors using a work-stealing algorithm [10, 17]. Section 2 presents the high level and low level programming interfaces. We show how the internal data flow representation is built at runtime with low

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PASCO'07, July 27–28, 2007, London, Ontario, Canada.
Copyright 2007 ACM 978-1-59593-741-4/07/0007 ...$5.00.

overhead. Moreover, we explain our lazy evaluation of readiness properties of the data flow graph following the work first principle of Cilk. This is similar to hybrid data flow architecture which allows to group together tasks for a sequential execution in order to reduce scheduling overhead. Nevertheless, KAAPI permits to reconsider at runtime group of tasks in order to extract more parallelism. Section 3 deals with the runtime system and the scheduling of KAAPI threads on the processors. The work-stealing algorithm relies on an extension of POSIX thread interface that allows work stealing. Next, we present the execution of data flow graph as an application of the general work-stealing algorithm when threads are structured as lists of tasks. This section ends with the presentation of the thread partitioning to distribute work among processors prior to the execution. The interaction with the communication sub-system to send messages over the network concludes this section. Section 4 reports experiments on multi-core / multi-processor architectures, clusters and grids up to 1400 processors.

2. THE KAAPI PROGRAMMING MODEL

This section describes the high level instructions used to express parallel execution as a dynamic data flow graph. Our instructions extend the ones proposed in Cilk [14] in order to take into account data dependencies. Following Jade [34], parallel computation is modelled from three concepts: tasks, shared objects and access specification. However, while Jade is restricted to iterative computations, here nested recursive parallelism is considered to take benefit from the work-stealing scheduling performances [8, 15, 17].

2.1 High level interface

The KAAPI interface is based on the Athapascan interface described in [15, 16]. The programming model is based on a global address space called *global memory* and allows to describe data dependencies between tasks that access objects in the global memory. The language extends C++ with two keywords[1]. The `shared` keyword is a type qualifier to declare objects in the global memory. The `fork` keyword creates a new task that may be executed in concurrence with other tasks. Figure 1 sketches the folk algorithm for computing the n-th Fibonacci number in a recursive way. A task is a function call: a function that should return no value except through the global memory and its effective parameters. The function signature should specify the mode of access (read, write or access) of the formal parameters in the global memory. For instance the function sum of the figure 1 is specified to have read accesses (`res1` and `res2`) and to have write access (`res`). The effective parameters in the global memory are passed by reference and other parameters are passed by value.

This very high level interface is very simple and only contains two keywords. There exists restrictions for passing effective parameter to formal parameter during task creation in order to ensure non-blocking execution of tasks. The reader should refer to [16] for a complete description of the interface and these restrictions.

[1]Currently this language extension uses C++ template definitions and specializations.

```
♯ include <athapascan-1>
void sum(shared_r int res1, shared_r int res2, shared_w int res)
{    res = res1+res2;    }

void fibonacci( int n, shared_w int res )
{
   if (n < 2) res = n;
   else {
      shared int res1;
      shared int res2;
      fork fibonacci(n-1, res1 );
      fork fibonacci(n-2, res2 );
      fork sum( res1, res2, res );
   }
}
```

Figure 1: Fibonacci algorithm with Athapascan interface. shared_w (resp. shared_r) means that the parameter is written (resp. read).

2.2 Low level interface and internal data flow graph representation

The KAAPI low level interface allows to build the macro data flow graph between tasks during the execution of Athapascan program. This interface defines several objects. A *closure* object represents a function call with a state. An *access* object represents the binding of an formal parameter in the global memory. The state of a closure can be *created*, *running*, *stolen* or *terminated* and represents the evolution of the closure during the execution. The state of the access object is one of the following: *created*, *ready*, *destroyed*. An access is created when a task is forked.

Once it is created, a closure should be bound to each effective parameter of the call. Binding of parameter by value makes copy. Binding of shared object in the global memory links together the last access to the object and the new access. The link between accesses is called *shared link* and represents the sequence of accesses on a same shared variable. Figure 2 illustrates the internal data flow graph after one execution of the task's body `fibonacci` of the figure 1. Let us note that creation of objects *closure* and *access* are pushed in a stack in order to improve allocation performance ($O(1)$ allocation time). Moreover, a stack is associated to each control flow, so that there is no contention on stack. When a closure is being executed, an activation frame is first pushed in the stack in order to restore the activation of the caller' stack environment after the execution. There is no *a priori* limit on the stack size: it is managed as a linked list of memory blocks of fixed size, except for some request. The system guarantees that an allocation request in a stack returns a contiguous block of memory, even at the expense of allocation of a block size by the system that corresponds to the request.

2.3 Semantic and natural execution order

The semantic is lexicographic: statements are lexicographically ordered by ';'. The value read from a parameter with **shared_r** access specification is the last written value according to the lexicographic order. This order is called the *reference order*.

This *reference order* is important for the execution: if only one processor is available for execution, then the execution

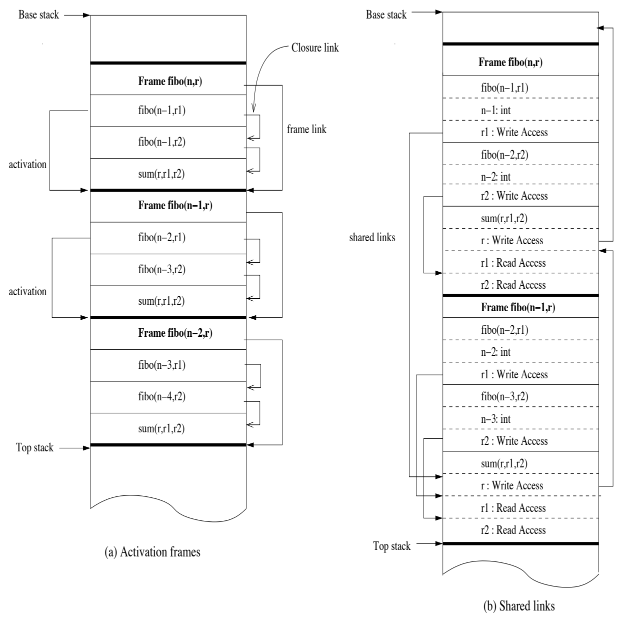


Figure 2: Data flow representation after one execution of a Fibonacci task.

of closures following this reference order is correct and does not require to compute the state of the closure objects and access objects of the data flow graph. This case corresponds to the standard execution, when no processor is idle.

Next section deals with scheduling when a processor becomes idle.

3. THREAD SCHEDULING IN KAAPI

The execution of a KAAPI program on a cluster is done by a dynamic set of processes, one per multi-processor, communicating through a network. Each process has several threads of control to execute tasks. In KAAPI a *thread* represents a computation. A thread could be an active control flow executing some computation or are inactive. At deployment step, one of the processes is designed to be the main process which starts the thread that executes the main task. This thread is called the *main thread* of the distributed execution.

Next sections present the thread interface and the non-preemptive scheduling algorithm by work stealing. Section 3.3 shows connection from the KAAPI thread to data flow computation.

3.1 Thread

Thread interface in KAAPI is a subset of POSIX threads [30] except for the functionalities about the management of signal and cleanup routines. We assume that readers are familiar with POSIX Thread and its interface.

The interface for thread management is closed to the POSIX threads: A thread may be created (`kaapi_create`), but not joined; mutex (`kaapi_mutex_t`) and condition (`kaapi_condition_t`) are the first class objects for thread synchronization.

KAAPI extends the POSIX interface in two ways. An extension of POSIX condition with additional integer value state is proposed. It allows threads to wait until a specific value is signaled: The thread is woken-up when the signaled value of the condition is equal (`kaapi_cond_wait_equal`) or not equal (`kaapi_cond_wait_notequal`) to the signaled value. This functionality is closed to one of those Futex provides [13].

The second extension concerns two new thread attributes. The first one is used to define functions called by the thread scheduler to extract work when one of the processors is idle, we named it the *steal function attribute*. The second attribute allows to specify two functions used to serialize thread through the network, it is named the *serialization attribute*. Threads with serialization attribute may be migrated[2] between processes (`kaapi_thread_migrate`) on some predefined points.

Figure 3 illustrates the creation of a KAAPI thread with the use of thread attribute. The steal function attribute

```
/* function that returns thief_thread that contains theft work
 * from the victim_thread. idle_cpu contains the set of idle
 * processors that initiate the call. In case of success,
 * function should return true.
 */
bool steal_function(
   kaapi_t victim_thread,
   kaapi_t* thief_thread,
   kaapi_cpu_t idle_cpu,
   void* user_arg
) { .. }

/* create a thread that may export work */
kaapi_attr_t attr;
kaapi_attr_init( &attr );
kaapi_attr_setstealfunction( &attr, &steal_function, ptr_to_arg );
kaapi_create( &tid, &attr, start_routine, arg );
```

Figure 3: Example of using KAAPI extension to create KAAPI thread with attribute to specify a way to extract work from existing thread.

parameters are: The first parameter is the KAAPI thread identifier on which the function is called; the second parameter is a pointer to a KAAPI thread identifier used to return a newly created thread that executes theft work; the third parameter contains the identifier of the idle virtual processor (also named kernel thread) that initiated the call. And the last parameter is a pointer to a user defined data structure. The return value is true if and only if a new thread has been created.

Next section presents the thread scheduling algorithm based on work-stealing.

3.2 Scheduling by work-stealing

The scheduling of KAAPI threads on the physical processors is a two level thread scheduling. The KAAPI runtime system schedules non-preemptively several user level threads on a fixed number of virtual processors (also called *kernel threads*). Virtual processors are scheduled by the kernel on physical processors. The KAAPI threads model is $M : N$, meaning that M user space threads are scheduled

[2]Note that the ability of defining this attribute, especially to migrate thread processor state (registers) is not of the responsibility of KAAPI.

onto N kernel threads [32]. This two thread scheduling levels take advantage of fast user space context switch between KAAPI threads while multi-processors are exploited with kernel threads.

Each KAAPI user level thread runs to completion, *i.e.* in a non-preemptive manner such as [19, 8, 15]. If a thread executes a blocking instruction, such as locking an already locked mutex or waiting for a condition variable, then it suspends: The kernel thread (or the virtual processor) is said *idle* and it switches to the *scheduler thread*. The scheduler thread is responsible to assign thread to idle virtual processor. The algorithm is simple. First, it tries to find a suspended thread ready for execution, like a standard thread scheduler. In case of success, the elected thread resumes its execution on the idle kernel thread. Else, the scheduler thread chooses at random a thread and call, if specified, the steal function attribute, as defined in previous section. If the function returns true then the newly created thread is scheduled on the kernel thread which becomes active.

On multi-cores / multi-processors, each core should has a kernel thread for scheduling work: the work-stealing scheduling algorithm is naturally distributed and the implementation has been carefully designed to avoid contention (for instance due to mutex). It shows good speedup even at fine grain (see section 4.1).

The scheduler thread is extended when group of processes execute the application. In case of failure, the kernel thread chooses at random a process and sends it request to steal work from its threads. On reception, the process trigger the request to try to steal work by calling the steal function attribute to several victim threads that have a serialization attribute. The reply message contains the output values, *i.e.* return value and the newly created thread. To improve locality of the work-stealing decision, severals steal operations on the local process are performed before emitting request to remote processes.

This thread scheduling principle by work-stealing is the kernel of implementation in [10], where the steal function attribute could be specialized to the work that is theft.

If a KAAPI thread performs blocking I/O, the kernel thread that executes it is also blocked [32]. In [2], kernel scheduler activation is introduced to improve performance of the user level application. The section 3.6 presents how KAAPI deals with communication between processes using non-blocking, one-side, active message communications. On Unix platform (mostly Linux and MacOSX), the implementation relies on user space context switch, either by using *makecontext/setcontext/ getcontext/ swapcontext*, or the more available interface *setjmp/ longjmp*. Kernel threads are implemented by the native thread library[3].

3.3 Using KAAPI thread for data flow computation

The high level API creates KAAPI threads that execute tasks following the reference order (section 2.3). Created threads have both the *steal function attribute* and the *serialization attribute* enabled. The steal function attribute is the most interesting function. It implements most of the work first principle of KAAPI design.

The thread changes the closure state to execute from *created* to *running*, pops it and executes it. This transition in the closure state diagram (figure 5) is atomic with respect to other transition, such as performed by the steal function call (see below). It is implemented using a compare and swap instruction. Moreover, it does not require the computation of the readiness of the closure thank to the reference order semantics. The overhead of executing a program with data flow representation is mainly due closure and access object creation cost.

If the transition failed due to the fact that initial state is *stolen* (the closure has been theft), then the running thread suspends its execution until the state of the closure becomes *terminated*. Then the scheduler thread is activated. The thread is suspended on our extension to POSIX condition described in section 3.1, waiting for the condition to be value '*terminated*'. On resume, the thread continues its execution following the reference order. When the closure completes, a new one is poped from the local deque. If this deque is empty, the thread terminates and the virtual processor switches to the scheduler thread, as explained in previous section. The steal function iterates through the bottom

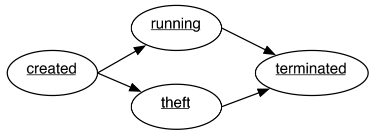


Figure 5: State diagram of closures. The two transitions from *created* state to *running* and *stolen* states are sources of contention between a victim thread executing its closures and a thief trying to steal it.

deque of the victim thread and tries to steal the first ready to execute closure not marked *running*. For each closure, the algorithm updates the state of all its accesses: If the access is marked *ready* and the immediate successor is an access with a concurrent mode of access to the global memory, then the successor is set to ready. If the mode of the access is write, the access is set to ready. When all accesses have been visited, and readiness propagated to successors, the closure is ready when all its accesses are ready.

The steal function attribute is called by the scheduler thread in case of inactivity. Once a ready victim closure is detected, the steal function tries to change its state from *created* to *stolen* (figure 5). If the transition is successful, then a new KAAPI thread is created with same attribute than the victim thread. The new thread contains an initial frame with a copy of the closure, in the state *created*, and a signalization closure with purpose is 1/ to transfer back the output data to the victim closure and 2/ to signal the completion of the victim closure, *i.e.* it moves its state from *stolen* to *terminated*. Figure 4 illustrates the stacks of both the victim thread and the thief thread after a successful attempt to steal a closure. The signalization closure is built with an exclusive access link after each access of the copied closure.

By the way a thread is waiting for completion of closure, the global computation is terminated when the main thread finishes to execute all its closures. Thus, the global terminaison detection does not need a distributed algorithm such as in [15]. It is implicit in our work-stealing implementation.

[3]For instance NPTL on recent Linux.

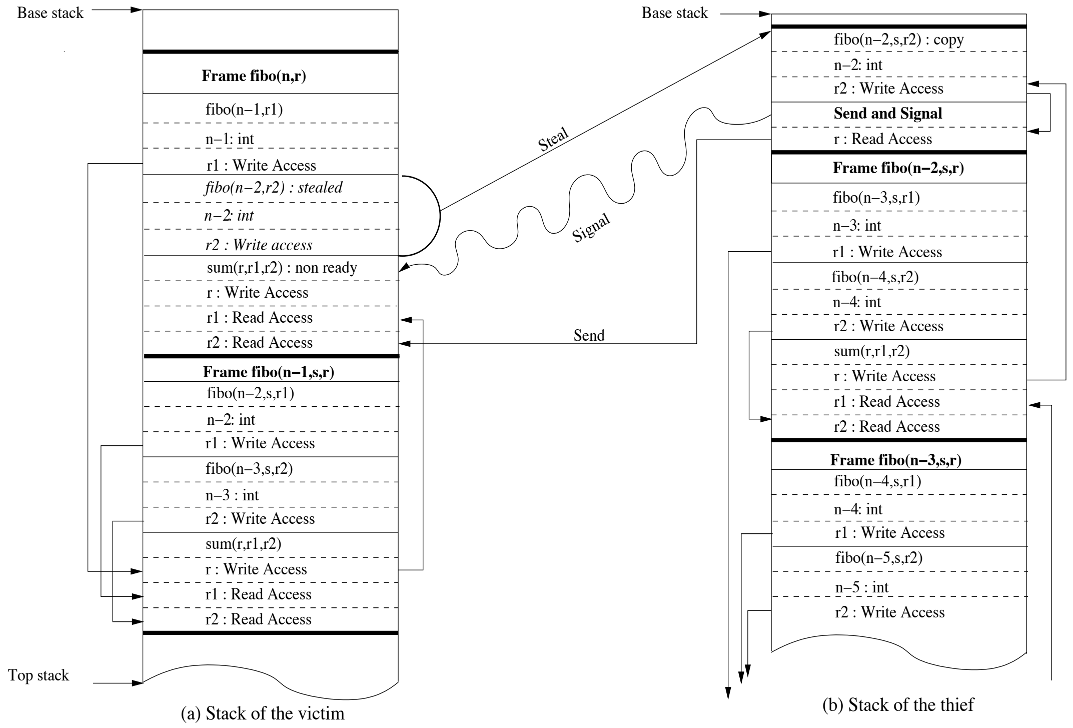


Figure 4: KAAPI Thread stacks after the steal operation. On the stack of the thief, the Send and Signal closure declares a read mode of access to the shared variable of the victim closure $Fibo(n-2, s, r2)$.

3.4 Performances of data flow computations scheduled by work-stealing

In [15, 17] a detailed analysis of this work-stealing algorithm is presented. We invite reader interested in the proof of these results to consult these publications. Performances on micro-benchmarks are presented in section 4.

3.5 Extension to thread partitioning

Work-stealing scheduling is efficient for multi-threaded computations [12, 5, 17]. Nevertheless, for iterative applications in numerical computations, the natural way is to describe the computation using loops. Even if algorithms may be rewritten using a recursive scheme to get benefit from work-stealing performances, from the point of view of the KAAPI runtime system it is natural to provide functionalities to programmers that enable their natural iterative description of the computations.

In KAAPI, we have designed a method to partition the data flow graph representing the work executed by a thread. Preliminary prototype has been partially published in [33]. The key point is to compute the schedule of tasks from the data flow graph: For each task, it computes the site of execution; and for each site, the order in which tasks are executed. The computation of a schedule is a plugin in KAAPI. Currently we have collected existing graph scheduling libraries, such as DSC [39], ETF [21] as well as libraries that partition data dependencies graphs (Metis [26] and Scotch [31]) which are efficient to solve mesh based problems occurring in most of scientific numerical simulations. These latter libraries are used in KAAPI to compute the data mappingby partitioning the sub graph of data dependencies from the data flow graph. Then, the site of execution of the tasks is deduced from the access they made to data in the global memory following an owner compute rule strategy [20].

Once the schedule is computed, the thread is partitioned in k threads, that form a group, which are distributed among the virtual processors on the processes. Communications between threads are based on added tasks in each thread: a task broadcast has a read access to data in the global memory and, on execution, it sends the value to destination; a task receive has a write access to data in the global memory and is terminated upon reception of value. Moreover, this enables to use the same work-stealing scheduling runtime support for managing communication as for executing multi-threaded computations.

Iterative computations reuse the group of threads by restoring their initial states. Redistribution of data occurs between iteration and are managed using our active message communication support.

3.6 Management of communication by work-stealing strategy

Communication in KAAPI are based on the active messages [27]. Sending a message is a non blocking operation. The sender process will be notified by a callback that data that compose message have been effectively sent. Upon reception, an user defined function is called by the KAAPI runtime system to process the message.

Non blocking sending is important and allows to overlap efficiently communication delays by computation. Moreover, in KAAPI, messages generated by a KAAPI thread are put in a private (local) queue. For each interface network card, a POSIX thread called *daemon*, is in charge of sending messages. The scheduling of sent messages is based on a work-stealing algorithm: the daemon indefinitely steals messages from the queues of KAAPI threads and sends them. Each steal operation tries to steal the largestt sequence of messages. This permits to aggregate messages together and decreases the transfer startup delay per message. Moreover, this kind of work-stealing for communication allows to hide network delay due to overloaded network: while the daemon is blocked during a network utilization, the KAAPI threads may continue to generate messages. Next time the daemon steals bigger messages sequences, thus aggregating more messages. Section 4.3 presents the capability of overlapping communication by computation.

4. EXPERIMENTS

We present here experimental results computed on the french heterogeneous Grid5000 [18] platform. Most of the current installed clusters are composed of dual-processors AMD. The most recent of them are dual-core dual-processors (AMD or Intel).

To deploy our software on the grid, we used the TakTuk [36] software. It allows us to replicate our environment on all sites (files copies and program compilation) and it offered us a parallel launcher of our application. This parallel launcher takes a list of machines (the machines we reserved), detects dead nodes, and deploy our KAAPI processes only on good nodes. This feature has been really useful as, each time we reserved hundreds of nodes, several of them were dead.

4.1 Fibonacci computation

A first set of experiments of the folk recursive Fibonacci number computation has been executed on a dual-core multi-processors machine (16 cores, AMD Opteron, 2.2Ghz and 32GBytes of main memory). Three versions of the benchmarks are compared: a pure C++, a version based on low level KAAPI interface and a version based on top of Athapascan API. We also compared two implementations of KAAPI.

In the timing results, we denote: T_1 the time, corresponding to the work of the parallel program, to execute the version KAAPI or Athapascan of the benchmark on one processor; T_p the time on p processors; T_S the time of the pure C++ sequential version of the benchmark.

	fib(35) ; $th = 2$			fib(40) ; $th = 2$		
p	T_p (second)	$\frac{T_1}{T_p}$	$\frac{T_1}{T_p^{old}}$	T_p (second)	$\frac{T_1}{T_p}$	$\frac{T_1}{T_p^{old}}$
1	0.99	1	1	10.75	1	1
2	0.49	1.96	1.90	5.65	1.98	1.96
4	0.26	3.91	3.61	2.80	3.99	3.90
8	0.13	7.57	5.38	1.41	7.96	7.38
16	0.084	11.81	4.67	0.73	15.33	10.30

Figure 6: Speedup on Fibonacci computation. Comparaizon between new implementation and old implementation of KAAPI thread scheduling.

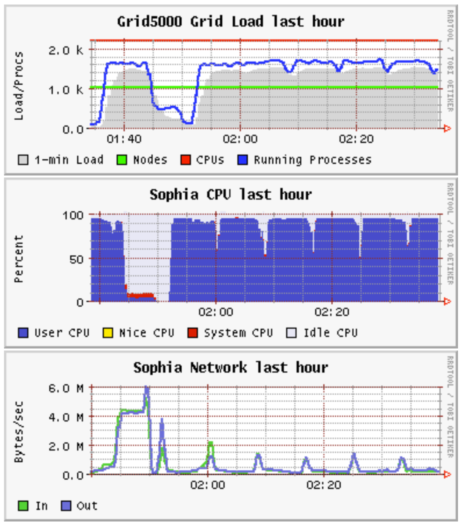
For the pure C++ sequential version, the time per function call is about $2.5ns$ (nano second), independent of the nth Fibonacci number to compute. Using low level interface of KAAPI, the measured overhead than the pure C++ is about $T_1/T_s = 13.4$, also constant with respect to the input n. The time per task (creation and execution) is about $33.5ns$. On top of Athapascan, the overhead is $T_1/T_s = 81$. This timing should be compared with Cilk related timing in [14] where an overhead nearly 3.6 is reported for the same benchmark. With KAAPI, more work is required to built data flow graph than Cilk implementation to enqueue task.

The table 6 presents timings for two implementations of KAAPI. Timings on two instances of Fibonacci are reported. The old implementation [17, 10] corresponds to the current stable version of KAAPI [25]. The new implementation is the one described in this paper and will be included in the new stable version of KAAPI.

Execution times on one processor for both implementations are equals. On the small instance, the sequential grain is about $0.33\mu s$, the maximum observed speedup with respect to parallel program on one processor is about 12 on 16 cores of the multi-processor: The new implementation is about 2.5 times faster than the old implementation. On the big instance, we observe a speed up of about 15.33 on 16 cores. The new implementation has a better scalability over the old one as the number of processors increases and the grain decreases. This is due to a finer implementation to reduce contention on internal data structure.

4.2 N-queens

We report here experimental results computed during the *plugtest*[4] international contest held in november 2006 at Sophia Antipolis, France. Our parallel NQueens implementation is based on the NQueens sequential code developped by Takaken [35]. It achieved the best performances, honored by a *special prize*[5] On instance 23 solved in $T = 4434.9s$, an idle time of $22.72s$ was measured on the 1422 processors; this

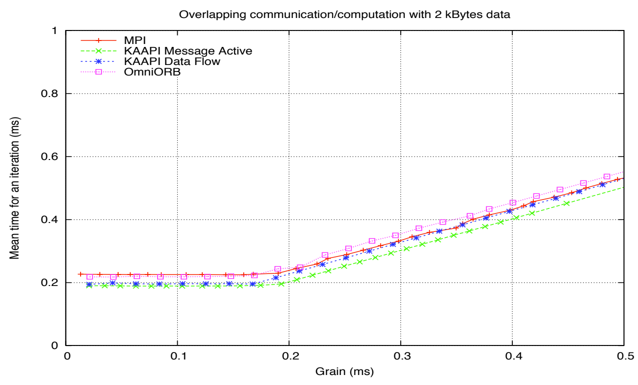


NQueens	p	T
21	1000	$78s$
22	1458	$502.9s$
23	1422	$4434.9s$

Figure 7: CPU/network loads and timing reports.

[4] http://www.etsi.org/plugtests/Upcoming/GRID2006/GRID2006.htm

[5] Final results are available on the ETSI url given above.



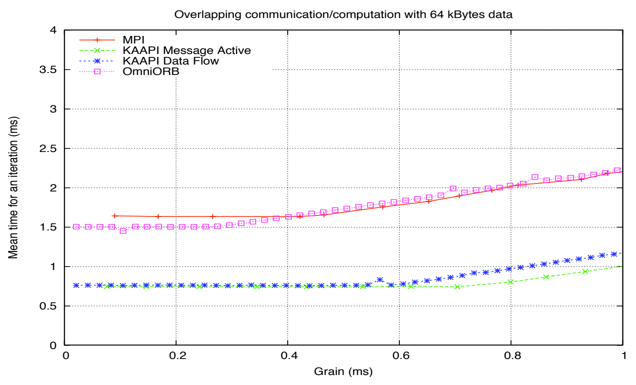


Figure 8: Measuring the overlapping capability of KAAPI. The upper graph reports timing for sending 2KBytes data with increasing local computation time. The lower graph is for sending 64KBytes data.

experimentally verifies our implementation efficiency with a maximal relative error $\frac{22.72}{4434.9} = 0.63\%$. Figure 7 shows the global grid load together with CPU and network load on one of the clusters composing the grid (cluster from the sophia site). These results have been obtained using Grid5000 monitoring tools during the last hour of execution. Our computations start approximately at 01:50. Different instances of nqueens problems are executed sequentially. The different graphs show a very good use of CPU resources. At the end of each execution work-stealing occurs, increasing briefly network load while enabling to maintain efficient CPU usage.

4.3 Measuring overlapping capability

In this experiment, we compare several middleware (MPI / MPICH, OmniORB, KAAPI Message Active, and Data Flow Computation with KAAPI) to measure the overlapping ratio on the following computation scheme on two processors communicating through TCP. The process P_0 initiates the communication of a message with size L to process P_1 using a non blocking send instruction; then P_0 computes for a fixed delay and waits an acknowledgment from P_1 upon reception of the message. The objectif is to measure the computation time over the total time ration which indicates how much the implementation is able to overlap communication delay (emission and reception) while computation is processed. On P_0, we have $T_{total} = T_{post} + T_{comp} + T_{wait}$, where each term represents the delays of the above described operations.

MPI implementation in this experiment uses `MPI_ISend` instruction to initiate the sending and posts the reception with `MPI_IRecv`. After the computation, the program waits the reception using `MPI_Wait`. OmniORB implementation relies on one-way CORBA method for non blocking communication. KAAPI Active Message implementation simply posts a message, performs computation and waits, using condition variable, the active message for the acknowledgment. The KAAPI data flow computation describes tasks such that the computation and communication are same as previous implementation. The data flow graph in the original program and the scheduling produce an execution on P_0 and P_1 such that:

- P_0 creates a task that produces data with the fixed L size, performs computation, then creates a task that wait for reading the acknowledgment data.
- P_1 creates a task that reads the data and writes an acknowledgment data (read by the last task on P_0).

The scheduling time, based on the METIS graph partitioning algorithm, for this instance is less than $10\mu s$ due to its relatively simplicity. Figure 8 reports timing for the four implementations. Until a certain threshold in the grain of the computation is reached, the total time is constant for both experiments: the total time is mainly due to the time to send message and receive acknowledgment. After this threshold, the total time is linear with respect to the computation which is on the critical path. The overlapping ratio is $\gamma = T^*_{cal}/T_{total}$, where T^*_{cal} is the grain of computation at this threshold. The first result is that the progress of communication in MPICH on top of TCP should imply the participation of the application and that OmniORB non blocking communication blocks the sender until the message was sent. The second result is that our active message implementation is efficient with respect to the concurrent progression of both communication and computation. Moreover, our thread partitioning scheme is able to take benefit of this overlapping capability.

4.4 Work-stealing versus thread partitioning scheduling on multi-processors

In this experience, we run a virtual simulation application on top of Sofa [1] comparing several parallel implementations. The objective is to have short delays in computation (around 30ms) in order to produce smooth image rate. The first implementation is based on pure work-stealing scheduling where a main thread generates few tasks that will themselves fork most of the computation.

Due to the short delay of the computation, the bottleneck in the scheduling is the access to the main thread by idle processors. Using thread partitioning technique, the first level of computation starts while workload has already been distributed. Figure 9 displays the speed up on a dual-core 8-processor machine (16 cores) for sequential computation time of about $30ms$. The iterative simulation is unrolled either on 10 loops or 20 loops.

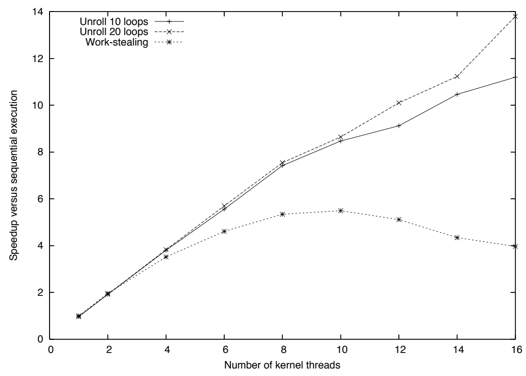


Figure 9: Thread partitioning versus work-stealing for Sofa application. Initial work distribution improves the dynamic work-stealing algorithm with increasing number of kernel threads.

5. CONCLUSIONS

Multithreaded computations may take benefit of the description of non strict data dependencies. In this paper we present the KAAPl approach to implement efficient multithreaded computations with data flow dependencies built at runtime. It is based on a small extension of POSIX thread interface and a basic work-stealing algorithm. Efficiency relies on the work-stealing algorithm as well as an original implementation directed by the work first principle that allows to report the cost to compute the state of the data flow graph on the critical path execution rather than on the work of the program. Experimental results show the feasibility of the approach. Work overhead is reasonably small and the work-stealing algorithm permits to reach good efficiencies for medium grain on both a multi-cores / multi-processors architecture and a PCs cluster. Moreover it scales on (heterogeneous) grid architecture as reported by the NQueens result.

Our thread partitioning approach, using graph scheduling and partitioning algorithms, is dedicated for iterative applications in numerical simulation. It is used for various applications: dynamic molecular *Tuktut*, cloth simulation *Sappe* [33] and *Sofa* [1]. Moreover, our past collaboration in the LinBox library [11] is under active development to produce several parallel algorithms, especially for a large sparse matrix-vector product [38] over finite field arithmetic. Thanks to the internal data flow representation, KAAPI maintains an abstract representation of the application execution. This representation has been used to develop original and efficient fault tolerance protocol with work-stealing [22, 23] or for iterative application [3] with application to adaptive computation [24].

Ongoing work is to target the KAAPI runtime on top an high speed network and managing internal data structure with affinity on the physical processors for NUMA architectures.

Acknowledgments

The author gratefully acknowledge Rémi Revire and Samir Jafar for useful discussions; Serge Guelton for participating to the implementation of the NQueens challenge.

6. REFERENCES

[1] J. Allard, S. Cotin, F. Faure, P.-J. Bensoussan, F. Poyer, C. Duriez, H. Delingette, and L. Grisoni. Sofa? an open source framework for medical simulation. In *Medicine Meets Virtual Reality (MMVR)*, 2007.

[2] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: effective kernel support for the user-level management of parallelism. In *SOSP '91: Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 95–109, New York, NY, USA, 1991. ACM Press.

[3] X. Besseron, S. Jafar, T. Gautier, and J.-L. Roch. Cck: An improved coordinated checkpoint/rollback protocol for dataflow applications in kaapi. In IEEE, editor, *Proceedings of the IEEE Conference on Information and Communication Technologies (ICTTA'06): from Theory to Applications*, pages 3353–3358, Damascus, Syria, April 2006.

[4] G.E. Blelloch. NESL: A Nested Data-Parallel Language. Technical Report CMU-CS-93-129, April 1993.

[5] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico.*, pages 356–368, November 1994.

[6] R. D. Blumofe and P. A. Lisiecki. Adaptive and reliable parallel computing on networks of workstations. In *Proceedings of the USENIX 1997 Annual Technical Conference on UNIX and Advanced Computing Systems*, pages 133–147, Anaheim, California, January 1997.

[7] R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[8] R.D. Blumofe and C.E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM Journal on Computing*, 1(27):202–229, 1997.

[9] D.E Culler and Arvind. Resource requirements of dataflow programs. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 141–150, Honolulu, Hawai, 1989.

[10] V. Danjean, R. Gillard, S. Guelton, J.-L. Roch, and T. Roche. Adaptive Loops with Kaapi on Multicore and Grid: Applications in Symmetric Cryptography. In *Proceedings of the Parallel Symbolic Computation (PASCO'07)*, 2007.

[11] J.-G. Dumas, T. Gautier, M. Giesbrecht, P. Giorgi, B. Hovinen, E. Kaltofen, B.D. Saunders, W.J. Turner, and G. Villard. Linbox: A generic library for exact linear algebra. In *Proceedings of the International Congress of Mathematical Software (ICMS'02), Beijing, China*, pages 40–50. World Scientific, 2002.

[12] P. Fatourou and P.G. Spirakis. Efficient scheduling of strict multithreaded computations. *Theory of Computing Systems*, 33(3):173–232, 2000.

[13] H. Franke, R. Russell, and M. Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in linux. In *Proceedings of the Ottawa Linux Symposium*, 2002.

[14] M. Frigo, C.E. Leiserson, and K.H. Randall. The implementation of the cilk-5 multithreaded language. In *Sigplan'98*, pages 212–223, 1998.

[15] F. Galilée, J.-L. Roch, G. Cavalheiro, and M. Doreille. Athapascan-1: On-line building data flow graph in a parallel language. In IEEE, editor, *Pact'98*, pages 88–95, Paris, France, October 1998.

[16] T. Gautier, R. Revire, and Roch. Athapascan: Api for asynchronous parallel programming. Technical Report RR-0276, APACHE, INRIA Rhône-Alpes, February 2003.

[17] T. Gautier, J.-L. Roch, and F. Wagner. Fine grain distributed implementation of a dataflow language with provable performances. In IEEE, editor, *Workshop PAPP 2007 - Practical Aspects of High-Level Parallel Programming in International Conference on Computational Science 2007 (ICCS2007)*, Beijing, China, May 2007.

[18] Grid5000. http://www.grid5000.org.

[19] L. J. Hendren, G. R. Gao, X. Tang, Y Zhu, X. Xue, H. Cai, and P. Ouellet. Compiling c for the earth multithreaded architecture. In IEEE, editor, *Pact'96*, pages 12–23, Boston, USA, 1996.

[20] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Houston, Tex., 1993.

[21] J.-J. Hwang, Y.-C. Chow, F. D. Anger, and C.-Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM J. Comput.*, 18(2):244–257, 1989.

[22] S. Jafar, T. Gautier, A. Krings, and J-L. Roch. A checkpoint/recovery model for heterogeneous dataflow computations using work-stealing. In *Proceedings of (LNCS) EuroPar'05*, Lisboa, Portugal, August 2005.

[23] S. Jafar, A. Krings, T. Gautier, and J-L. Roch. Theft-induced checkpointing for reconfigurable dataflow applications. In *Proceedings of the IEEE Electro/Information Technology Conference EIT2005*, Lincoln, Nebraska,U.S.A., May 2005.

[24] S. Jafar, L. Pigeon, T. Gautier, and J.-L. Roch. Self-adaptation of parallel applications in heterogeneous and dynamic architectures. In IEEE, editor, *Proceedings of the IEEE Conference on Information and Communication Technologies (ICTTA'06): from Theory to Applications*, pages 3347–3352, Damascus, Syria, April 2006.

[25] Kaapi. http://kaapi.gforge.inria.fr.

[26] G. Karypis and V. Kumar. Analysis of multilevel graph partitioning. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 29, New York, NY, USA, 1995. ACM Press.

[27] A. Mainwaring and D. Culler. Active message applications programming interface and communication subsystem organization. Technical Report CSD-96-918.

[28] C. J. Morrone, J. N. Amaral, G. Tremblay, and G. R. Gao. A Multi-Threaded Runtime System for a Multi-Processor/Multi-Node Cluster. In Kluwer Academic, editor, *15th Annual International Symposium on High Performance Computing Systems and Applications*, pages 18–20, Windsor, ON, Canada, 2001.

[29] G.J. Narlikar. Scheduling threads for low space requirement and good locality. Number TR CMU-CS-99-121, may 1999. Extended version of the paper published in Spaa'99.

[30] Institute of Electrical and Inc. Electronic Engineers. Information Technology — Portable Operating Systems Interface (POSIX) — Part: System Application Program Interface (API) — Amendment 2: Threads Extension [C Language]. IEEE Standard 1003.1c–1995, IEEE, New York, NY, 1995.

[31] F. Pellegrini and J. Roman. Experimental analysis of the dual recursive bipartitioning algorithm for static mapping. Technical Report 1038-96, 1996.

[32] M. L. Powell, Steve R. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks. Sunos multi-thread architecture. In *USENIX Winter*, pages 65–80, 1991.

[33] R. Revire, F. Zara, and T. Gautier. Efficient and easy parallel implementation of large numerical simulation. In Springer, editor, *Proceedings of ParSim03 of EuroPVM/MPI03*, pages 663–666, Venice, Italy, 2003.

[34] M.C. Rinard and M.S. Lam. The design, implementation, and evaluation of Jade. *ACM Trans. Programming Languages and Systems*, 20(3):483–545, 1998.

[35] Takaken. http://www.ic-net.or.jp/home/takaken/e/queen.

[36] TakTuk. http://taktuk.gforge.inria.fr.

[37] X. Tang, J. Wang, K. B. Theobald, and G. R. Gao. Thread partitioning and scheduling based on cost model. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 272–281, 1997.

[38] R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. *Journal of Physics Conference Series*, 16:521–530, January 2005.

[39] T. Yang and A. Gerasoulis. DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors. *IEEE Trans. Parallel Distrib. Syst.*, 5(9):951–967, 1994.